Topics

- Bytecode verification
- Limiting code behavior
  - By origin
  - By signature
  - Who is asking — JAAS sketch
- Authentication and authorization in J2EE web applications
  - BASIC and FORM login
  - Encrypting server communications
Topics

- Key and certificate management
  - Creating keys
  - Obtaining certificates
  - Being your own CA
  - Certificate revocation

- Java Secure Sockets Extension (JSSE)
  - Controlling cipher suites
  - Anonymous SSL
  - SSL with server authentication
  - SSL with mutual authentication
Bytecode Verification
Bytecode Verification

- One.java

```java
public class One {
    public static void main(String[] args) {
        Two t = new Two();
        int x = t.getX();
        System.out.println("Got an int " + x);
    }
}
```

- Two.java

```java
public class Two {
    public int getX() {
        return 42;
    }
}
```
Bytecode Verification

- Disassemble classes using `javap -c`

```java
% javap -c Two
Compiled from "Two.j"
public class Two extends java.lang.Object{
    public Two();
    Code:
        0:    aload_0
        1:    invokespecial  #2;  //Method java/lang/Object."<init>":()V
        4:    return

    public int getX();
    Code:
        0:    bipush  24
        2:    ireturn
}
```
Assemble a “Broken” Class

- Two.j (Note: jasmin source != javap output)

```
.class public Two
.super java/lang/Object

; standard initializer
.method public <init>()V
   aload_0
   invokenonvirtual java/lang/Object/<init>()V
   return
.end method

.method public getX()I
   bipush 24
   lreturn ; BAD, attempt to return long from int type method
.end method
```

- Assembles OK:

```
% java -jar ~/jasmin-1.1/jasmin.jar Two.j
Generated: Two.class
```
Verification Fails

- Running this is not permitted:

```java
% java One
Exception in thread "main" java.lang.VerifyError: (class: Two, method: getX signature: ()I) Expecting to find long on stack
    at One.main(One.java:3)
```
Symbolic References

- Refs.java

```java
public class Refs {
    private int x;
    private float y;
    private Object obj;
    public int getX() {
        return x;
    }
    public float getY() {
        return y;
    }
    public Object getObj() {
        return obj;
    }
}
```
Symbolic References

- javap -c Refs

Compiled from "Refs.java"
public class Refs extends java.lang.Object{
  public Refs();
    Code:
    0:  aload_0
    1:  invokespecial #1; //Method java/lang/Object."<init>":()V
    4:  return
  public int getX();
    Code:
    0:  aload_0
    1:  getfield #2; //Field x:I
    4:  ireturn
  public float getY();
    Code:
    0:  aload_0
    1:  getfield #3; //Field y:F
    4:  freturn
  public java.lang.Object getObj();
    Code:
    0:  aload_0
    1:  getfield #4; //Field obj:Ljava/lang/Object;
    4:  areturn
}
Limiting Code Abilities
Controlling Code Behavior

- Cat.java

```java
import java.io.*;
public class Cat {
    public static void main(String [] args) throws Throwable {
        FileInputStream fis = new FileInputStream(args[0]);
        FileOutputStream fos = new FileOutputStream(args[1]);
        int c;
        while ((c = fis.read()) != -1) {
            fos.write(c);
        }
    }
}
```

- What will happen (retrain the programmer? :)

```
java Cat Cat.dat Dog.dat
```

- What about:

```
java -Djava.security.manager Cat.dat Dog.dat
```
Security Manager Defaults

- Java takes "default deny" stance (mostly)

```java
% java -Djava.security.manager Cat Cat.dat Dog
Exception in thread "main"
java.security.AccessControlException: access denied (java.io.FilePermission Dog write)
  at
  java.security.AccessControlContext.checkPermission(AccessControlContext.java:264)
  at
  ...
  at java.io.FileOutputStream.<init>
(FileOutputStream.java:70)
  at Cat.main(Cat.java:6)
```
Security Manager Control

- `${JAVA_HOME}/jre/lib/java.policy`
  
  ```
  // Standard extensions get all permissions by default
  grant codeBase "file:${java.ext.dirs}/*" {
    permission java.security.AllPermission;
  };
  
  // Default permissions granted to all domains
  grant {
    permission java.lang.RuntimePermission "stopThread";
    permission java.net.SocketPermission "localhost:1024-", "listen";
    permission java.util.PropertyPermission "java.version", "read";
    permission java.util.PropertyPermission "java.vendor", "read";
    permission java.util.PropertyPermission "java.vendor.url", "read";
    ...
  }

  - Note, no special permission for local code unless in `java.ext.dirs`

  - Can force use of a specific policy file:
    ```
    java -Djava.security.manager \
    -Djava.security.policy=./myjava.policy ...
    ```
Adding a Permission

- Insert into (local) policy file:

```java
grant codebase
   "file:/home/simon/java/Security/secman/" {
   permission java.io.FilePermission
   "/home/simon/java/Security/secman/-", "read";
   permission java.io.FilePermission
   "/home/simon/java/Security/secman/-", "write";

   Grants privilege to code loaded from /
   home/simon/java/Security/secman

   To read and write files under that same
directory
```
Effect of Policy Entry

- **Now:**
  
  ```java
  java -Djava.security.manager 
  -Djava.security.policy=~/java.policy 
  Cat ../Cat.dat Dog.dat
  ```

- **fails, but:**
  
  ```java
  java -Djava.security.manager 
  -Djava.security.policy=~/java.policy 
  Cat Cat.dat Dog.dat
  ```

- **succeeds**
**JAR Signatures Control Behavior**

- **Create key pair under (alias “orinoco”)**
  - keytool -genkey -alias orinoco
  - Answer CN questions

- **Put class in a jar**
  - jar cf Cat.jar Cat.class

- **Sign the jar**
  - jarsigner Cat.jar orinoco

- **Optionally verify it:**
  - jarsigner -verify -verbose -certs Cat.jar
Run the Code from the Jar

- Install security manager, but not special policy
  
  ```
  java -Djava.security.manager -classpath ./Cat.jar Cat ../Cat.dat x.dat
  ```
  
  Fails (expected)
Grant Needed Permissions

- Grant file read/write permission to code signed by orinoco

- Add to policy file:

```java
keystore "./keystore", "jks";
...
grant signedBy "orinoco" {
    permission java.io.FilePermission "/home/simon/java/Security/secman/-", "read";
    permission java.io.FilePermission "/home/simon/java/Security/secman/-", "write";
};
```
Run with Permissions

- Execute code with permissions from policy:

```java
java -Djava.security.manager \
    -Djava.security.policy=./java.policy.sig \
    -classpath ./Cat.jar Cat Cat.dat Dog.dat
```

- This time it works
Security Manager Behavior

- Potentially sensitive calls are passed through the SecurityManager/AccessController

- Examines the stack:
  - Caller's protection domain
  - Protection domains on the thread's stack
  - Method arguments
  - Which user (principal) is running the code

- (JAAS provides tools for user login and attaching a Principal to a thread)
Authentication and Authorization in J2EE Web Applications
Securing Web Applications

- Goals are:
  - Prove who is making a request (Authentication)
  - Control who can do what (Authorization)

- Two easy steps:
  - Make the users log in
  - Control what they can do based on who they are
Making Users Log In

- Deployment descriptor (web.xml) specifies:
  - Security constraint, containing:
    - Web resource collections
      - List of URLs
    - Authorization constraints
      - List of user “roles”
  - A login mechanism
  - Enumeration of known roles
Key `web.xml` Elements

```xml
<security-constraint>
  <web-resource-collection>
    <web-resource-name>Secure</web-resource-name>
    <url-pattern>/*</url-pattern>
  </web-resource-collection>
  <auth-constraint>
    <role-name>staffmember</role-name>
  </auth-constraint>
</security-constraint>

<login-config>
  <auth-method>BASIC</auth-method>
  <realm-name>basic-file</realm-name>
</login-config>

<security-role>
  <role-name>staffmember</role-name>
</security-role>
```

**What?**
- `<security-constraint>`
- `<web-resource-collection>`
  - `<web-resource-name>`Secure`<web-resource-name>`
  - `<url-pattern>`/*`<url-pattern>`
- `<auth-constraint>`
  - `<role-name>`staffmember`<role-name>`

**Who?**
- `<security-role>`
  - `<role-name>`staffmember`<role-name>`

**How?**
- `<login-config>`
  - `<auth-method>`BASIC`<auth-method>`
  - `<realm-name>`basic-file`<realm-name>`

**Who's possible?**
- `<security-role>`
  - `<role-name>`staffmember`<role-name>`
Roles and Role Mapping

- **“Users” is a platform dependent notion**
  - Platform specific mechanism lists valid users
    - (and credentials, such as passwords)

- **J2EE uses “Role” as a platform independent notion**

- **Roles are mapped to users in deployment**
  - Using server specific configuration file
  - `sun-web.xml` for AS8/RI
Role Mapping in AS8

- In `sun-web.xml` file:

  ```xml
  <sun-web-app>
    <security-role-mapping>
      <role-name>staffmember</role-name>
      <principal-name>j2ee</principal-name>
      <group-name>staff</group-name>
    </security-role-mapping>
  </sun-web-app>
  ```

- Any of “j2ee” or “staff” will satisfy a request for a “staffmember”
Creating and Managing Users

- User credentials are platform specific

- In AS8, create a user:
  
  ```
  asadmin create-file-user --user admin
  [--groups staff] new_user
  ```

- And set a password:
  
  ```
  asadmin update-file-user --user admin
  --userpassword passwd new_user
  ```
Login Mechanisms

- BASIC
- FORM
- DIGEST
- CLIENT-CERT

- FORM is the usual choice
  - Needs configuration
  - Needs login and error pages
  - Needs transport security
Specifying Forms

- Form authentication requires custom login and error pages

  ➢ Configuration is in web.xml:

```xml
<login-config>
  <auth-method>FORM</auth-method>
  <realm-name>form-realm</realm-name>
  <form-login-config>
    <form-login-page>
      /login.jsp
    </form-login-page>
    <form-error-page>
      /error.jsp
    </form-error-page>
  </form-login-config>
</login-config>
```
Form Elements

- Login form must collect username and password

- Standard (mandatory) form elements are:
  - ACTION="j_security_check"
  - INPUT TYPE="text" NAME="j_username"
  - INPUT TYPE="password" NAME="j_password"
Securing Communications

- Form login data is unsecured by default
- Must mandate SSL on login page (and all subsequent pages)
- Add “transport guarantee” to security constraint

```xml
<security-constraint>
  <user-data-constraint>
    <transport-guarantee>
      CONFIDENTIAL
    </transport-guarantee>
  </user-data-constraint>
</security-constraint>
```
Transport Guarantees

- CONFIDENTIAL is usual
  - Ensures encryption, using server certificate

- INTEGRAL is also possible
  - Ensures data not tampered with
  - But not secret

- Once you're using SSL, keep with it
  - Use an intercepting filter pattern
Keys and Certificates

- AS8 requires its keys/certificate under alias “s1as”
  - Other appservers probably have their own expectations
- Must provide key and certificate for CONFIDENTIAL transport
- Certificate name should match host URL or browser will issue a warning
Managing Keys and Certificates in Java
Keys and Certificates in Java

- `keytool` does most manipulation for you
  - alias is a handle/local name
    - Should match any application expectations
    - e.g. AS8 uses “s1as”
  - Common Name (CN)

- Certificate formats
  - Binary, keytool's default
  - Text, use `-rfc` flag
    - Commonly used in Internet situations, e.g. signing requests (mailable)
Storing Keys

- Default is a file-based store
  - Certificates are encrypted using a password
  - Private keys are doubly encrypted using a second password
  - Protect the file!
- Certificates contain a public key
- Keystores can contain private keys too
  - For your own keys
  - Java concept of keystore and truststore
Create Your Keys

- Create a key pair in keystore:
  - `keytool -genkey -alias s1as -keystore keystore.jks`

- Enter keystore password
  - Default for AS8 and examples is “changeit”
  - Also a private key password, can be the same

- Enter CN elements
  - Use FQDN as first/lastname for a webserver
Obtaining a Certificate (Option 1)

- Two keys are not a certificate
- Create a certificate signing request:
  - `keytool -certreq -alias s1as -rfc`
- Send request to a CA (e.g. Verisign)
  - Include money and proof
- Get a certificate back
- Import the certificate alongside your private key entry
Importing Certificates

- Import your new signed certificate
  - Use the same alias
  - Alias and public key must match

- Import a CA certificate
  - Error if alias exists
  - Be sure certificate is sound first!
  - Use -printcert option to check details
Be Your Own CA

- Internal certificates don't have to cost that much
- keytool, openssl, and certtool can do this for you
  - Keytool manipulates Java's keystore/truststore
  - certtool generates keys you can extract
    - keytool keeps private key private
  - openssl x509 command manipulates and signs
  - Watch for the use of text (pem) format
  - (Other tools/combinations will work too)
Create the CA Keys

- Create a CA private key
  - `certtool --generate-privkey --outfile CA.pem`

- Create signing request from this
  - `certtool --generate-request --load-privkey CA.pem --outfile CAreq.pem`

- Now self-sign the request
  - `openssl x509 -req -in CAreq.pem -signkey CA.pem -out CACert.pem`
Trust the CA Certificate

- Java must be told to trust this CA (self-signed) certificate
  
  -keytool -import -v -trustcacerts -alias CA -rfc -file CAcert.pem

- Now, anything signed by our CA will be trusted too
  
  - Even if not pre-loaded into our keystore/truststore
Use Your CA to Sign Certificates

- Export certificate request to `req.pem`
  - `keytool -certreq -alias s1as -rfc -file req.pem`

- Sign the certificate using your CA keys
  - (CAcert.pem and CA.pem created earlier)
  - `openssl x509 -req -in req.pem -CA CAcert.pem -CAkey CA.pem -out cert.pem`

- Then import resulting certificate (`cert.pem`)
  - `keytool -import -rfc -alias s1as -file cert.pem`
Certificate Authority Flow

- Create your own C.A. key pair
- Create self-signed certificate (CA certificate)
- Install CA certificate as trusted “throughout” your company
- Use CA private keys to sign internal certificates
  - Be sure to protect the private keys! (CA.pem in this example)
Being a Certificate Authority

- Cheaper
- Gives you control
- Appropriate inside a company
  - Perhaps between two cooperating companies
Certificate Revocation

- Java ignores CRLs by default
- In-house certificates might not need CRLs, external ones probably do
- Enable revocation lists in security configuration file

```
${JAVA_HOME}/lib/jre/security/java.security
ocsp.enable=true
```
Java Secure Sockets Extension
Private Communications
Code Sequence with JSSE

- Import the classes
- Obtain a server socket factory
- Create the server socket
- Accept the client connection
- Optionally, configure the system
  - Must be before the handshake
  - Specify cipher suites
  - Require client authentication
SSL Handshake

- Notice the TCP connection is accepted before the start of SSL handshake
- Handshake, or any transmission from server completes the session
Code Essentials with JSSE

- Core api elements always needed are:
  - `import javax.net.ssl.*;`
  - `SSLSocketFactory.getDefault`
  - `createServerSocket(port)`
  - `accept`
  - `getInputStream`, `getOutputStream`
Example 1, What Ciphers?

```java
import java.io.*; import java.net.*; import javax.net.ssl.*;

public class ServerOne {
    public static void main(String[] args) {
        SSLServerSocketFactory sf = (SSLServerSocketFactory) SSLServerSocketFactory.getDefault();
        SSLServerSocket ss = null;

        try {
            ss = (SSLServerSocket) sf.createServerSocket(7777);
        } catch (Exception ex) { ex.printStackTrace(); }

        System.out.println("Supported Ciphers:");
        String[] ciphers = sf.getSupportedCipherSuites();
        for (String s: ciphers) { System.out.println(s); }

        System.out.println("Enabled Ciphers:");
        ciphers = ss.getEnabledCipherSuites();
        for (String s: ciphers) { System.out.println(s); }
    }
}
```
Example 2, These Ciphers!

- Server Side:

```java
ss = (SSLServerSocket)sf.createServerSocket(7777);
String [] ciphers = ss.getEnabledCipherSuites();
String [] enable = new String[ciphers.length + 1];
System.arraycopy(ciphers, 0, enable, 0, ciphers.length);
enable[ciphers.length] = "TLS_DH_anon_WITH_AES_128_CBC_SHA";
ss.setEnabledCipherSuites(enable);

Socket sock = ss.accept(); // can get SSLSocket using cast
PrintWriter pw = new PrintWriter(sock.getOutputStream());
pw.println("Hello, this is a secret handshake");
pw.flush();
pw.close();
sock.close();
```
Example 2, These Ciphers!

- **Client Side:**

```java
SSLContext sf = (SSLContext)SSLContext.getDefault();

try {
    SSLSocket sock = (SSLSocket)sf.createSocket(
        args[0], new Integer(args[1]));

    String[] ciphers = sock.getEnabledCipherSuites();
    String[] enable = new String[ciphers.length + 1];
    System.arraycopy(ciphers, 0, enable, 0, ciphers.length);
    enable[ciphers.length] = "TLS_DH_anon_WITH_AES_128_CBC_SHA";
    sock.setEnabledCipherSuites(enable);

    BufferedReader br = new BufferedReader(
        new InputStreamReader(sock.getInputStream()));
    String sin;
    while ((sin = br.readLine()) != null) {
        System.out.println("Received: "+ sin);
    }
    sock.close();
} catch (Exception ex) {
    ex.printStackTrace();
}
```
Anonymous SSL

- No certificates required
- Talk with high security, but with whom?
- Not a good idea, but demonstrates:
  - Control of cipher suite
  - Need for bilateral agreement
  - Easy code example: pure Java
Example 3, psst...

- Server (requires certificate/private key):

```java
ss = (SSLServerSocket)sf.createServerSocket(7777);

Socket sock = ss.accept(); // can get SSLSocket using cast
PrintWriter pw = new PrintWriter(sock.getOutputStream());
pw.println("Hello, this is a secret handshake");
pw.flush();
pw.close();
sock.close();
```

- Client (requires trusted CA certificate):

```java
SSLSocket sock = (SSLSocket)sf.createSocket(
    args[0], new Integer(args[1]));
BufferedReader br = new BufferedReader(
    new InputStreamReader(sock.getInputStream()));
String sin;
while ((sin = br.readLine()) != null) {
    System.out.println("Received: " + sin);
}
sock.close();
```
Providing Keys for JSSE

- SSL requires certificates and keys
- Override the defaults using properties:
  - -Djavax.net.ssl.trustStore=<path>
  - -Djavax.net.ssl.trustStorePassword=<pwd>
  - -Djavax.net.ssl.keyStore=<path>
  - -Djavax.net.ssl.keyStorePassword=<pwd>
Debugging SSL

- SSL handshake is non-trivial
- Failures can be hard to track down
- Enable debug output with:
  -Djavax.net.ssl.debug=ssl,handshake
- Shows (extensive!) information describing:
  - keystore & truststore used
  - keys found and trust applied
  - Cipher suites available and chosen cipher
  - Session key in use
Example 4, Who's There?

Client side:

```java
SSLSSocket sock = (SSLSSocket) sf.createSocket(
    args[0], new Integer(args[1]));
// Get the session (also triggers the handshake)
SSLSSession session = sock.getSession();
try {
    System.out.println("Got a session with Principal "+
        session.getPeerPrincipal());
} catch (SSLPeerUnverifiedException pue) {
    System.out.println("Unverified remote Principal ");
}
BufferedReader br = new BufferedReader(
    new InputStreamReader(sock.getInputStream()));
String sin;
while ((sin = br.readLine()) != null) {
    System.out.println("Received: "+ sin);
}
sock.close();
```
Example 5, Prove Yourself Client!

- Server side (requires installed CA cert verifying client):

```java
SSLRequest sock = (SSLRequest) ss.accept();
sock.setNeedClientAuth(true); //before handshake starts
sock.startHandshake(); //not really necessary
SSLSession session = (SSLSession) sock.getSession();
try {
    System.out.println("Got a session with Principal " +
                      session.getPeerPrincipal());
} catch (SSLPeerUnverifiedException pue) {
    System.out.println("Unverified remote Principal ");
}
```

- Client side requires installed certificate and private key
Key API Review
Java Security APIs

- SecurityManager/AccessController
- Permissions, Signatures, Message Digests — java.security
- JAAS — javax.security.auth.*
- JGSS — org.ietf.jgss
- JSSE — javax.net.ssl
- keytool — key management
- certtool, openssl x509 — 3rd party C.A. tools